UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

IMPLEMENTACIÓN DE FRONTEND EN REACT PARA MAGENTO 2

Jesús Menéndez Cuesta
Tutor: Jorge Carretie Romero
Ponente: Alejandro Sierra Urrecho

JUNIO 2019

IMPLEMENTACIÓN DE FRONTEND EN REACT PARA MAGENTO 2

AUTOR: Jesús Menéndez Cuesta
TUTOR: Jorge Carretie Romero

Dpto. Ingeniería del Software
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio de 2019

Resumen

El objetivo de este Trabajo Fin de Grado consiste en realizar un *frontend* para Magento 2 en React. Actualmente el *frontend* de Magento 2 hace uso de ficheros XML, PHTML, HTML, LESS y JS. Este *frontend* carece de velocidad de respuesta y aunque se puede optimizar usando una cache como Varnish, la experiencia de navegación en dispositivos móviles no es la mejor.

Para la implementación de este nuevo *frontend* se hará uso del proyecto de código abierto PWA Studio de Magento. Este proyecto nos da una buena base para empezar, ya que con ello nos aseguramos de que seguimos él mismo planteamiento que el oficial. De esta manera podemos aprovecharnos del desarrollo futuro realizado tanto por Magento como por la comunidad. El *frontend* de React incluido, denominado "Venia concept", se hará extensible de manera que sea lo más parecido al sistema actual de temas de Magento 2 y sea así aprovechable para diferentes proyectos sin tener que partir de cero.

Para las comunicaciones entre el *frontend* y el *backend* el proyecto PWA Studio utiliza GraphQL, un lenguaje que nos permite obtener datos del *backend* sin tener que recurrir a la API REST de Magento 2 de manera que cada módulo de Magento 2 puede establecer los métodos, parámetros y la respuesta de manera sencilla.

Finalmente, para demostrar el potencial del desarrollo realizado, se utilizará este *frontend* para realizar la tienda online de un cliente real, y así poder comparar la velocidad ganada en comparación a un *frontend* estándar de Magento 2.

Abstract

The purpose of this Bachelor Thesis is to develop a frontend for Magento 2 in React. At the moment, Magento 2 frontend's uses XML, PHTML, HTML, LESS and JS files. This frontend lacks response speed and, despite it can be optimized using a cache like Varnish, the mobile web browsing experience is not the best.

For the implementation of this new frontend we will make use of the open source project called PWA Studio of Magento. This project will be a solid foundation to start our project, since with that we will make sure to follow the same approach as the official one. This way we can take advantage of the future developments made by Magento and its community as well. The React frontend included, called "Venia concept", will be made extensible as possible to the actual theme system of Magento 2 so it can be reused for different projects without starting from scratch.

To communicate the frontend with the backend PWA Studio project uses GraphQL, a language that allow us to obtain data from the backend without using the default API REST of Magento 2. This way every Magento 2 module can establish the methods, parameters and response in an easy way.

Finally, to show the potential of the development done, this extensible frontend will be used to perform a real online store of a client, comparing the speed and advantages over the standard Magento 2 frontend.

Palabras clave

Magento, React, Frontend, GraphQL, Proyecto de código abierto

Keywords

Magento, React, Frontend, GraphQL, Open source project

Agradecimientos

A mi madre Isabel, a mi padre Jesús y a mi hermana Vanesa

INDICE DE CONTENIDOS

INDICE DE FIGURAS

INDICE DE TABLAS

1 Introducción

1.1 Motivación

Esta memoria de TFG trata sobre la implementación de un *frontend* basado en React para el *framework* de comercio electrónico Magento.

Magento es un *framework* escrito en PHP y es una de las principales plataformas en este ámbito. Su gran adaptabilidad lo convierte en uno de los mejores *backends* para gestionar cualquier tipo de comercio electrónico, desde pequeños a grandes negocios. Además de la gran cantidad de extensiones, tanto de código abierto como de pago, Magento ofrece una gran cantidad de recursos para que los desarrolladores puedan adaptar Magento a las necesidades del cliente.

Las grandes ventajas que tiene su *backend* son reducidas por el *frontend* de este *framework*. Aunque su sistema de temas es una ventaja y permite a los desarrolladores modificar cualquier aspecto del *frontend*, incluso pudiendo definir varios temas que se pueden cambiar en simples pasos desde el *backend*, su sistema de ficheros, falta de documentación para ciertas partes del *frontend* y gran cantidad de módulos hacen que este *framework* sea algo complejo para desarrolladores que desconocen su utilización y aplicación, como fue mi caso al realizar las prácticas en mi actual empresa.

Para realizar un *frontend* en Magento es muy probable que tengamos que modificar o crear archivos HTML, CSS o LESS, JavaScript, PHTML y XML. No siempre es una tarea sencilla saber cuáles de los módulos estándar de Magento están introduciendo ciertos elementos en una página. Además, estos ficheros no pueden tener la estructura que queramos, ya que tienen que seguir una cierta jerarquía la cual se muestra en el siguiente esquema:

```
<theme_dir>/
├── <Vendor>_<Module>/
│   ├── web/
│   │   ├── css/
│   │   │   ├── source/
│   ├── layout/
│   │   ├── override/
│   ├── templates/
├── etc/
├── i18n/
├── media/
├── web/
│   ├── css/
│   │   ├── source/
│   ├── fonts/
│   ├── images/
│   ├── js/
├── composer.json
├── registration.php
├── theme.xml (Referencia [1])
```

Esta jerarquía y la gran cantidad de ficheros que se utilizan en el *frontend* dificultan el desarrollo para alguien sin experiencia previa en el *framework*. Además, su velocidad de carga es algo lenta y, sin optimizaciones, puede llegar a ser bastante pesado para el cliente y el servidor. Es muy posible que, sin utilizar una compresión como por ejemplo de la Gulp, se generen paquetes JS de hasta 7 MB, algo muy pesado para plataformas móviles. Teniendo en cuenta que según los análisis de datos de páginas webs que maneja la empresa puede suponer de media el 70% del tráfico, agilizar la navegación para esta plataforma es algo necesario.

Debido a este problema, nace la idea de utilizar Magento como *backend*, ya que éste tiene una gran cantidad de APIs REST para comunicarse con el *frontend* de la web y poder realizar todo tipo de operaciones, y comunicarlo con un *frontend* totalmente independiente del actual que para hacerlo rápido y más fácil de modular. Debido a los buenos resultados obtenidos en otros desarrollos similares, se elige ReactJS para dicho cometido.

ReactJS es una biblioteca en JavaScript de código abierto desarrollada por Facebook diseñada para crear diferentes interfaces de usuario. Este tipo de tecnología es declarativa ya que está basada en componentes, lo cual facilita de manera significativa la tarea de diseño al poder declarar las vistas y los distintos estados de la aplicación de una manera intuitiva.

Además, otra de las grandes ventajas de React es que con cada cambio de estado de uno los componentes solo se renderizan los componentes afectados por dicho cambio, produciéndose este mismo de manera muy rápida entre estados o vistas de la aplicación. Este factor será muy ventajoso en la navegación en dispositivos móviles, donde el *frontend* de Magento se nota algo más lento, y además debido al uso del renderizado por parte del cliente se pueden ahorrar costes a la hora de desplegar la aplicación en servidores.

Como idea inicial, se valoró la posibilidad de usar las APIs REST de Magento para comunicar el *backend* y el *frontend* utilizando React para este último, para así obtener lo mejor de cada desarrollo. Investigando sobre esta posibilidad, se da con el "PWA Studio" de Magento, el cual es un proyecto de código abierto impulsado por desarrolladores independientes y de Magento. Este proyecto usa un *frontend* en React y proporciona todas las herramientas necesarias para comunicarlo con el *backend* de Magento. Debido a la importancia que creemos que tendrá este desarrollo para el futuro de la plataforma, decidimos basarnos en este para la realización de este Trabajo de Fin de Grado.

1.2 Objetivos

- Desarrollar un frontend extensible basado en React utilizando el proyecto PWA Studio de Mageno como base.
- Utilizar el desarrollo para realizar el frontend de una tienda real y demostrar sus posibilidades.
- Adaptación de ciertos módulos actuales de Magento para su correcto funcionamiento en el nuevo frontend.
- Comparar la velocidad, tanto en ordenador como en dispositivos móviles, del frontend actual de Magento con el desarrollado para este TFG.

1.3 Organización de la memoria

La memoria consta de los siguientes capítulos:

- Estado del arte
- Diseño
- Desarrollo
- Integración, pruebas y resultados
- Conclusiones y trabajo futuro

2 Estado del arte

2.1 Proyecto PWA Studio

El ejemplo principal de tecnología similar a la que queremos implementar es el proyecto del que parte nuestro desarrollo, el PWA Studio de Magento.

El proyecto PWA Studio de Magento tiene como objetivo el desarrollo de una *progressive web app* o aplicación web progresiva. Los objetivos de esta aplicación son el de crear sitios web que sean rápidos, seguros, adaptables, con compatibilidad multi-navegador, accesibles incluso sin conexión, gracias a los *Service Workers* que se encargan de ejecutar ciertas acciones en segundo plano para obtener información o notificaciones *push,* y que puedan ser utilizados en los móviles como aplicaciones para poder recibir notificaciones o compartir contenido a través de redes sociales o distintas aplicaciones instaladas en el terminal. Referencia [2]

El proyecto PWA Studio cuenta con un *frontend* desarrollado en React denominado "Venia concept" que tiene como objetivo mostrar el potencial de las herramientas aportadas para la realización de PWAs compatibles con Magento y el cual será la base para el desarrollo de nuestro trabajo de fin de grado.

2.2 ScandiPWA

Otro ejemplo de desarrollo de una aplicación PWA de Magento es ScandiPWA. Está desarrollada usando herramientas muy similares a las de PWA Studio, como React, GraphQL y promete tiempos de carga hasta 3 veces superiores, un ratio de conversión mayor en dispositivos móviles, así como la posibilidad de subir aplicaciones dedicadas para Android e iOS, todo esto sin requerir ningún *middleware* adicional.

El diseño del frontend de ScandiPWA está orientado a móvil, al igual que el de PWA Studio, y es que como ya hemos mencionado anteriormente gran parte del tráfico de internet hoy en día se realiza a través de dispositivos móviles, pudiendo probar las posibilidades de este desarrollo en la siguiente url : https://demo.scandipwa.com/

3 Diseño

3.1 Estructura del proyecto

Este proyecto consta de dos partes bien diferenciadas y común en aplicaciones de React, el *frontend* manejado por este último y un *backend* comunicado a través de distintas soluciones, en este caso se utiliza las API Rest estándares de Magento para ciertos casos, como el carrito o la información del *checkout*, y GraphQl para otras comunicaciones, como proveer de información a las distintas categorías o productos.

La instancia de Magento se usará en modo *headless*, es decir, el *frontend* de esté no será utilizado, por lo que se podría usar una máquina o instancia mejor optimizada para este tipo de uso, al igual que se podría hacer con el frontend en React.

En todos los proyectos partimos de una carpeta *htdocs* en la que dentro está el *framework* que se utilizaría, normalmente Magento, pero al estar dividida en dos partes, decidimos separarlas en dos partes bien diferenciadas. En la carpeta "magento-backend" se encuentra el *framework* Magento 2.3.1 y en la carpeta "magento-frontend" se encuentra el proyecto PWA Studio en su versión 2.1.0. A continuación, vamos a detallar la estructura del *backend* y el *frontend*.

3.1.1 Magento

La estructura de Magento 2 consta de varias carpetas que, utilizadas de manera correcta, diferencian las distintas partes de la aplicación. En concreto vamos a detallar como creemos que se debe utilizar esta estructura para tener un *backend* bien modulado.

Para empezar, la carpeta "app" contiene gran parte del código que los desarrolladores pueden añadir a Magento. Dentro de esta carpeta encontramos "code" la cuál puede contener los módulos de desarrolladores, pero que en nuestro caso utilizamos para código específico que es específico de la tienda y que no puede ser reutilizado para otros proyectos. Por ejemplo, podríamos tener un módulo genérico que genere productos a través de un archivo CSV pero los atributos de los productos no serían iguales de una tienda a otra, por lo que en esta carpeta "code" introduciríamos la personalización requerida.

Otra carpeta dentro de "app" es "design", donde se introduciría el tema para el *frontend* de un Magento estándar. En este proyecto se omite por completo esta carpeta, debido a que no es necesaria al introducir el *frontend* en React. Por lo tanto, no entraremos en muchos más detalles.

Por último, siempre dentro de la carpeta "app" encontramos la carpeta "etc" donde se encuentran, entre otros, el archivo "env.php". Éste contiene las variables de entorno necesarias para el funcionamiento de Magento, como son conexión a la base de datos, configuración de cache y Redis, entre otras. También encontramos el archivo "config.php", que contiene el estado de activación de los módulos que componen Magento.

Una vez vista la estructura y archivos más importantes de la carpeta "app", encontramos varias carpetas más al mismo nivel, destacando la carpeta que denominamos "extensions". Esta carpeta no forma parte de un Magento base, pero es donde creemos que deberían estar los módulos que pueden ser compartidos de un proyecto a otro, independientemente de los

requisitos de estos, y tiene que ser definida como repositorio en el archivo "composer.json", que se encuentra en la raíz del *framework* para poder los módulos contenidos en esta carpeta como dependencia del proyecto con el comando "composer require".

Dentro seguimos la estructura de nombramiento de carpetas que sigue Magento, que empieza con "vendor", correspondiente al nombre del desarrollador del módulo, dentro de esta "module" cuyo nombre suele ser "module-<nombre_del_módulo>" y dentro la estructura de un módulo estándar de Magento.

Esta estructura básica suele componerse de la siguiente estructura:

- **Carpeta "Api"**: donde se pueden definir interfaces que luego implementarán otras clases.
- **Carpeta "Block"**: donde están definidas las funciones que son utilizadas por los bloques que componen el *frontend*, tanto de la tienda como del administrador de Magento.
- **Carpeta "Console"**: aquí se definen los comandos de consola accesibles a través del comando "php bin/magento".
- **Carpeta "Controller"**: contiene los controladores que el desarrollador puede definir para una cierta ruta, tanto para el *frontend* de la tienda como para el administrador.
- **Carpeta "etc"**: contiene la versión de datos y esquema que define el módulo, las inyecciones de dependencia que pudiera tener el módulo o la configuración que se puede asignar desde el administrador.
- **Carpeta "Helper"**: en ella se suelen definir los "helpers", clases que contienen funciones que son utilizadas por varias clases, para evitar la duplicidad de código.
- **Carpeta "i18n"**: contiene las traducciones necesarias para los textos del módulo.
- **Carpeta "Model"**: normalmente contiene las definiciones de los modelos de base de datos que son necesarios para el módulo.
- **Carpeta "Setup"**: contiene la definición del esquema de base de datos. Puede declararse un fichero que solo definirá el esquema inicial, pudiendo declarar en otro fichero las sucesivas actualizaciones correspondientes a la versión del módulo.
- **Carpeta "view"**: en esta carpeta se encuentran todos los ficheros que definen los elementos visuales tanto del *frontend* como del administrador.
- **Fichero "composer.json"**: necesario para poder crear la dependencia de composer.
- **Fichero "registration.php"**: este fichero contiene el nombre del módulo que será utilizado entre otras cosas para activarlo en el fichero "config.php" del que se hablo anteriormente en la carpeta "app".

Con la versión 2.3 de Magento se añadió la posibilidad de extraer información del *backend* a través de sentencias "query" de GraphQL. Este tema se tratará más en profundidad en la sección siguiente.

Por último mencionar la carpeta “vendor” que contiene los archivos del núcleo de Magento al igual que todos los módulos necesarios definidos como dependencia en el fichero “composer.json” junto con los enlaces simbólicos desde la carpeta “extensions” de los módulos contenidos por esta.

3.1.2 GraphQL

Desde la versión 2.3 de Magento se permite el uso de GraphQl para obtener información del *backend* de Magento. GraphQl es un lenguaje para requerir y manipular datos, siendo más flexible, potente y eficiente que una API REST tradicional. La ventaja que tiene GraphQL es que no depende del sistema de base de datos y puede adaptarse con facilidad al código y los datos ya existentes, pudiendo definir los tipos de datos que necesitamos sin ningún tipo de restricción.

El diseño por el que ha optado el equipo de Magento es el de separar los módulos, teniendo una parte para el *framework* como hasta ahora y un módulo adicional para manejar las peticiones de GraphQL. Por ejemplo, para el módulo de clientes tenemos el “module-costumer”, el cual ya existía, y el “module-costumer-graph-ql”. Este diseño permite adaptar módulos existentes para que pueden ser compatibles, como veremos más adelante.

Un módulo de Magento que utiliza GraphQL se compone de los archivos necesarios para su activación como módulo, como ya hemos visto en la sección anterior, además de un archivo “schema.graphqls” dentro de la carpeta “etc” que contiene las definiciones de tipos de datos y *queries* que serán utilizadas para obtener datos. Un ejemplo de archivo de definición de estructura de GraphQL sencillo sería el siguiente:

```
type Query {
        customers(
                pageSize; Int
                currentPage: Int
        ): [Customer]
        @ resolver(class:”Ezenit\\Graphql\\Model\\Resolver\\Customer”)
}

type Customer {
        name: String,
        email: String,
        phone: String
}
```

Este archivo definiría un *query*, la cual podemos requerir a través del nombre “customers” y tendría dos parámetros opcionales de tipo Int que corresponden al tamaño de página que queremos recuperar y la página en la que estamos pudiendo, por ejemplo, paginar de manera muy sencilla una tabla de clientes. Los parámetros son opcionales ya que no llevan el carácter “!” después del tipo. Esta *query* devuelve un array de tipo “Customer” el cual devuelve un objeto con los campos con los tipos definidos en él.

Una vez definido el esquema necesitamos una clase que se ocupe de recuperar y devolver los datos de la manera especificada, siendo esta la clase denominada “resolver” en la

definición del tipo *query*. Esta clase tiene una función "resolve" la cual recibe los parámetros de entrada y devuelve, en este caso, un array con la estructura definida en el tipo que devuelve la *query*.

Este ejemplo se puede probar con la extensión de Chrome denominada "ChromeiQL" la cual nos permite establecer una ruta, en este caso la ruta de Magento encargada de responder peticiones de GraphQL, y lanzar las *querys* pudiendo visualizar la respuesta de las mismas, obteniendo el siguiente resultado:

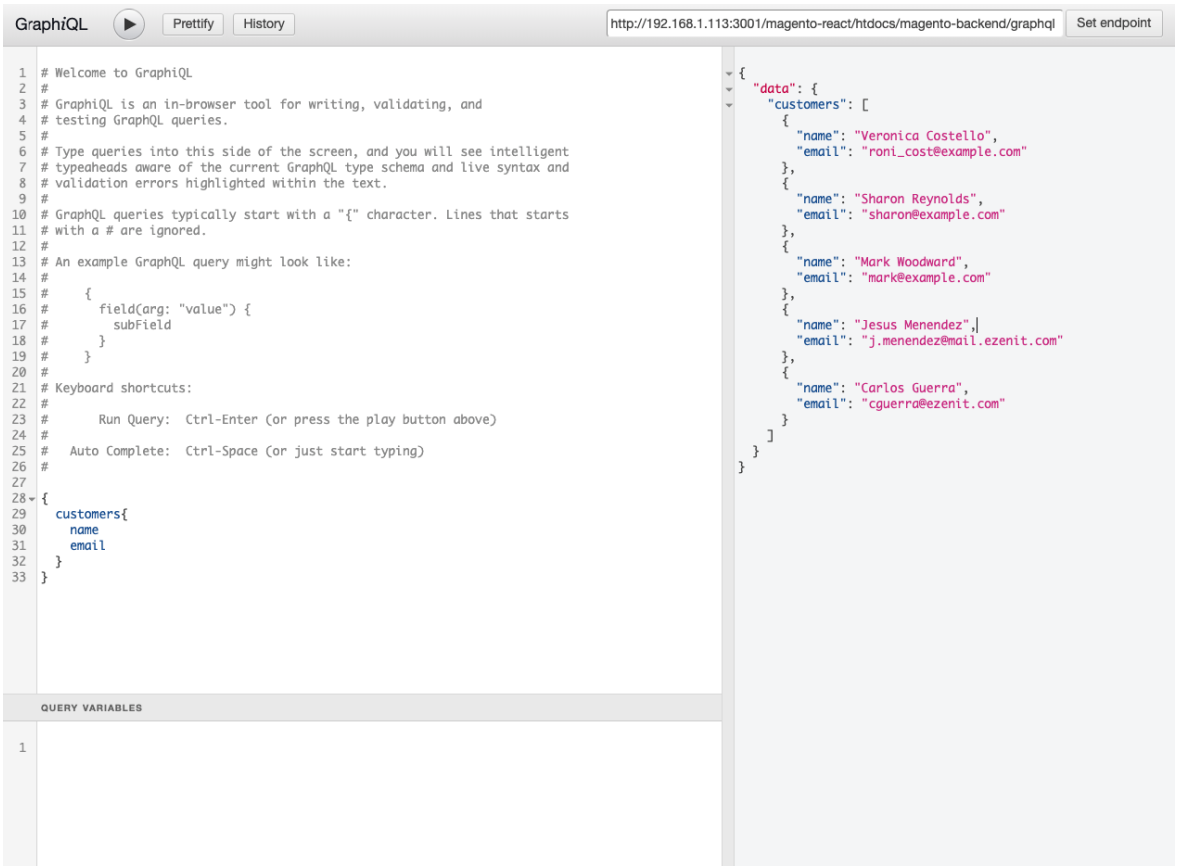

Figura 3-1: Resultado de la query "customers" en ChromeiQL

Para lanzar la *query* en el editor simplemente la requerimos por el nombre y entre corchetes ponemos el nombre de los campos que queremos recuperar, no siendo necesario poner todos los campos del tipo que devuelve la *query* si no solo los que necesitamos, siendo en este caso ignorado el teléfono de los clientes.

3.1.3 PWA Studio

El proyecto PWA Studio, y en concreto el paquete "venia-concept" serán los encargados de manejar el *frontend* de nuestra aplicación. El proyecto PWA Studio tiene como objetivo el proveer a los desarrolladores con herramientas con las que puedan construir aplicaciones web progresivas utilizando Magento como *backend* y se compone de varios paquetes manejados por el gestor de paquetes Yarn.

Estos paquetes están definidos como espacios de trabajo los cuales tienen distintas dependencias. Según la documentación del proyecto tenemos:

- **Paquete pwa-buildpack**: que contiene las principales herramientas de construcción y desarrollo para una PWA de Magento.
- **Paquete peregrine**: contiene una colección de elementos de interfaz de usuario para una PWA de Magento.
- **Paquete venia-concept**: una prueba de concepto de construcción de una frontend para Magento 2 usando las herramientas de PWA Studio.

Referencia [3]

La estructura del paquete venia-concept es muy similar a la de otras aplicaciones de React, donde destaca la carpeta "src" que contiene todos los distintos componentes que forman el *frontend*, la carpeta "queries" donde se encuentran las definiciones de queries que están relacionadas con las definidas en el *backend* como hemos visto en la sección anterior, y la carpeta "RootComponents" que contiene los componentes raíz equivalentes a las distintas vistas de páginas que podemos tener, como la página de categoría, la de producto o la de búsqueda.

El *frontend* venia concept puede ser accedido desde el siguiente enlace, aunque hay que tener en cuenta que la velocidad es lenta debido a que el servidor en el que está alojada no es muy potente: https://magento-venia.now.sh/

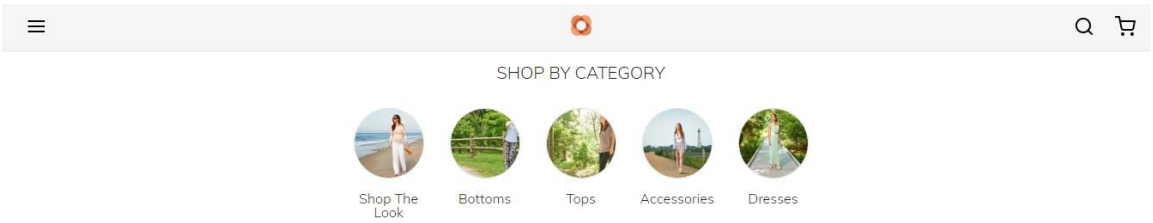

Figura 3-2: Página de inicio de Venia Concept

El *frontend* Venia Concept nos ofrece una vista muy simplificada, con un diseño orientado a móvil, en el que el menú, carrito y búsqueda son elementos que se despliegan al hacer click sobre ellos.

La página de aterrizaje, *landing page*, nos ofrece un enlace para cada una de las categorías recuperadas desde el *backend* y el *footer*, el cual contiene un diseño muy simple.

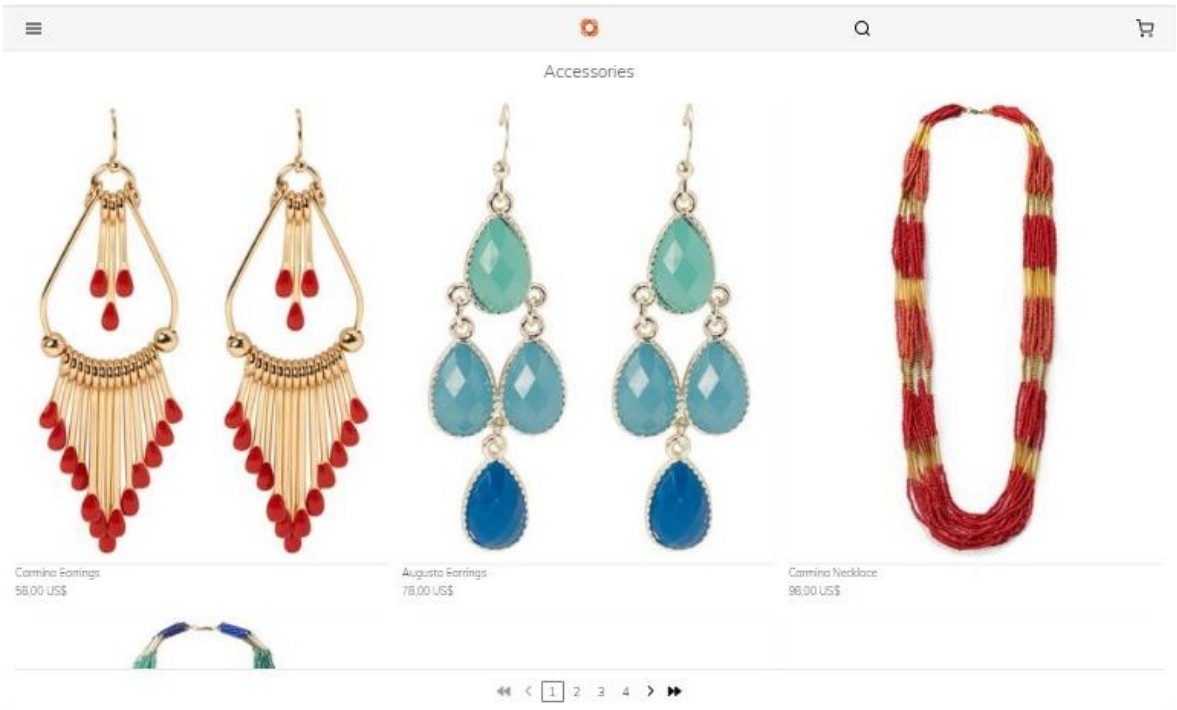

Figura 3-3: Vista de categoría de Venia Concept

La simplicidad del *frontend* continua en la vista de categoría, encontrándonos con un listado de productos con elementos de gran tamaño y la información básica del producto, con su nombre y precio. También está implementada la paginación, pudiendo movernos por las distintas páginas.

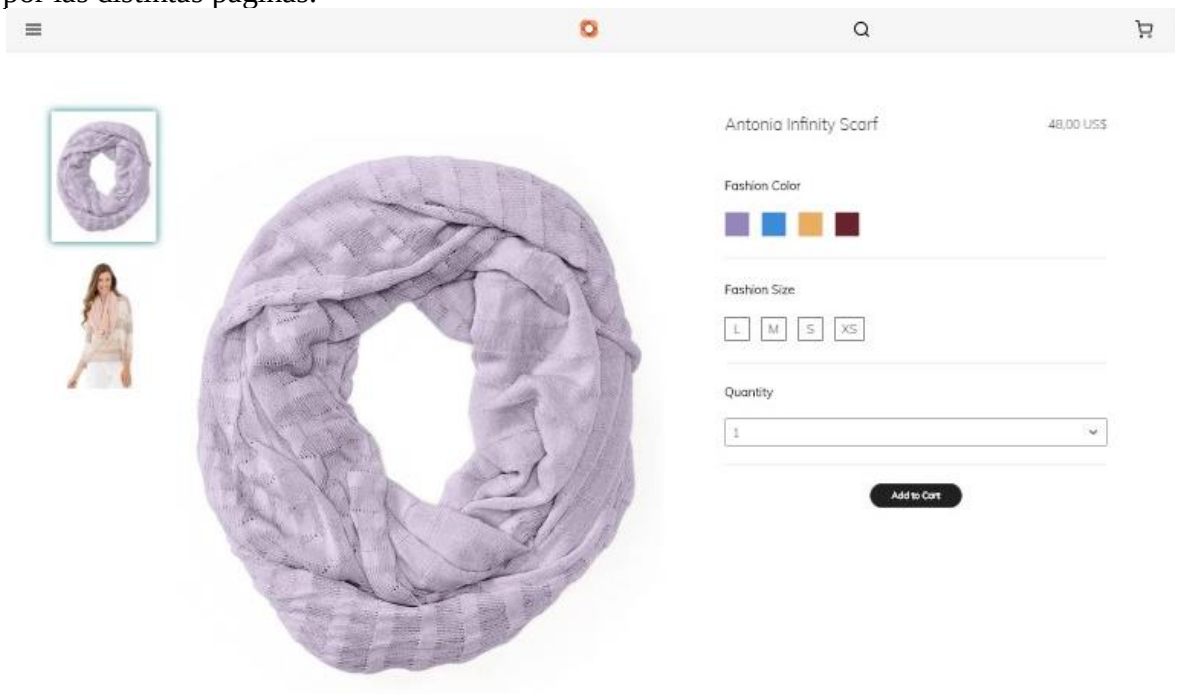

Figura 3-4: Vista de producto de Venia Concept

La vista de producto cuenta con la posibilidad seleccionar las opciones del producto, ya sea de color, talla y cantidad además de una galería con vistas en miniatura y una imagen de mayor tamaño, algo más abajo se encuentra la descripción del producto y el código del artículo.

El *checkout* está integrado en carrito, pudiendo realizar la compra en 4 simples pasos y de manera muy rápida.

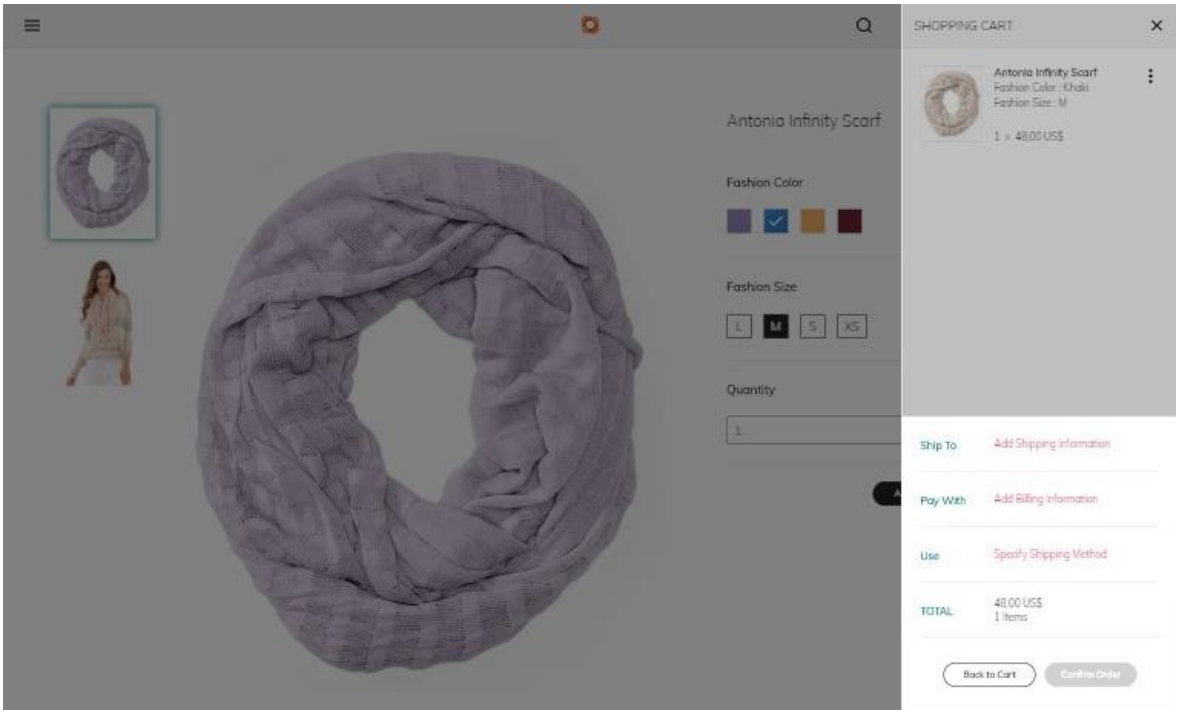

Figura 3-5: *Checkout* de Venia Concept

4 Desarrollo

4.1 Extensibilidad

Desde que encontramos el proyecto PWA Studio nos preguntamos si podríamos desarrollar algo parecido al sistema de temas de Magento, en el que todo parte de un tema base, incluido en el *framework*, modificándolo para adaptarlo al diseño requerido por el cliente. Esta extensibilidad en Magento se consigue a través de la carpeta "design" donde podemos registrar un tema y modificar todas y cada una de las vistas de los módulos que componen el *frontend*.

Con el proyecto PWA Studio ya teníamos un *frontend* básico con ciertas funcionalidades, como el carrito, el *checkout* o la búsqueda, que podíamos reutilizar sin problemas, mientras que otras partes, como el *header* o el *footer*, necesitarían de personalización para adaptarse a diferentes diseños. Como referencia para este proyecto utilizaremos el diseño de la página de un cliente, en concreto la página http://www.sierrabravacarp.com/, a la cual se intentará ser lo más fiel posible introduciendo algunas mejoras aportadas por el nuevo *frontend*.

Con esto en mente empezamos a investigar el proyecto en busca de una forma con la que poder conseguir la extensibilidad de los componentes ya implementados por parte del equipo de desarrollo de Magento.

El proyecto hace uso de un paquete para React denominado "CSS-Loader", el cuál se encarga de que dentro de un componente de React se pueda importar un archivo CSS para usar sus clases y en concreto hace uso de "CSS modules", el cual permite que las clases declaradas e importadas por un componente solo se apliquen para este. Esto tiene la ventaja de que las clases y declaraciones que hagamos para un componente no se apliquen para otro, facilitando el uso de CSS y evitar el uso de LESS.

Los módulos de CSS se pueden personalizar, y en la configuración se especifica con que nombre se deben exportar las clases para que sean únicas para cada módulo. Es por esto que todos los componentes de "venia-concept" se exportan a través de una función llamada "classify".

Esta función se encarga de coger el nombre del componente y asignarle las clases correspondientes de manera que estas se apliquen con un identificador único. Además, esta función realiza una lógica antes de devolver el componente para que sea renderizado por React, lo cual nos da la opción de poder realizar la extensibilidad que buscábamos al principio. El código que insertaremos en archivo "classify" será el siguiente:

```
import React, { Component } from 'react';

import getDisplayName from 'src/util/getDisplayName';

import * as extensions from 'src/extensions';

const merge = (...args) => Object.assign({}, ...args);

const classify = defaultClasses => WrappedComponent =>
```

```
  class extends Component {
    static displayName = `Classify(${getDisplayName(WrappedComponent)})`;

    checkExtensions(){
      var originalComponent = getDisplayName(WrappedComponent);
      if(originalComponent in extensions.MAP){
        var extendedComponent = extensions.MAP[originalComponent];
        WrappedComponent = extensions[extendedComponent](WrappedComponent);
        defaultClasses   = merge(defaultClasses, WrappedComponent.defaultClasses);
      }
    }

    render() {
      {this.checkExtensions()}
      const { className, classes, ...restProps } = this.props;
      const classNameAsObject = className ? { root: className } : null;
      const finalClasses = merge(
        defaultClasses,
        classNameAsObject,
        classes
      );
      return <WrappedComponent {...restProps} classes={finalClasses} />;
    }
  };

export default classify;
```

Lo primero que hará la función “render” del fichero “classify” modificado será buscar extensiones a través de la función *checkExtensions*. Esta función se encargará de buscar con el nombre del componente original si hay alguna extensión que lo sustituya y, de haberla, sustituir el componente por el extendido. Esta correspondencia entre componente original y extensor se consigue a través del archivo index.js que se importa de la carpeta “extensions”, la cual es añadida a la estructura ya existente. Este archivo consiste en lo siguiente:

```
import EzHeader              from './ezenit/module-header';
import EzLogo                from './ezenit/module-logo';

var MAP = {
  "Logo"             : "EzLogo",
  "Header"           : "EzHeader"
}

export {
  EzLogo,
  EzHeader,
  MAP
};
```

Este archivo se encarga de exportar los componentes que extienden, evitando tener todos los “imports” en el archivo “classify”, junto con una variable “MAP” que contiene el

mapeo entre componente original y su extensor. Con todo esto conseguiríamos devolver un componente extensor en el render, pero es necesario definir como deberían ser estos componentes.

En un principio pensamos que podría ser posible hacer que nuestros componentes extendieran los de Magento, estableciendo una herencia, para luego sobrescribir los métodos que hiciera falta, y aunque hubiera sido lo ideal para lo que tratamos de conseguir, no fue posible debido a una restricción de React que establece que los componentes solo pueden extender la clase "React.Component". Referencia [4]

Para ello lo primero era decidir si íbamos a optar por sustituir el componente por completo o si íbamos a sustituir solo algunas funciones de este, como el render, para alterar la forma en la que se muestra el componente o ciertas partes de su funcionamiento según nuestros criterios. Finalmente optamos por sustituir ciertas funciones exportando nuestros módulos a través de una función que se encargaría de sustituir solo las funciones sobrescritas por el componente, de manera que fuera posible solo alterar la funcionalidad de este dependiendo de las necesidades

Dentro de la carpeta "extensions", previamente definida, se encontrarían todos los componentes extendidos, teniendo un impacto mínimo en la estructura y funcionamiento del proyecto ya existente de manera que sea compatible con futuras actualizaciones lanzadas por Magento.

La función que exportaría nuestros componentes y se encargaría de sustituir los métodos estaría en la raíz de nuestra carpeta "extensions" en el archivo denominado "plugin.js". Este archivo contendría el siguiente código:

```
import _ from 'lodash';

function pluginBuilder(ExtendedComponent, DefaultClasses){
  return function(OriginalComponent){
    var WrappedComponent = ExtendedComponent;

    var original_methods = Object.getOwnPropertyNames(OriginalComponent.prototype);
    var extended_methods =
Object.getOwnPropertyNames(ExtendedComponent.prototype);

    if(extended_methods.length == 1){
      OriginalComponent.defaultClasses = DefaultClasses;

      return OriginalComponent;
    }
    for (var i = 0; i < original_methods.length; i++) {
      if(original_methods[i] !== "constructor"){
        if( _.includes(extended_methods, original_methods[i]) ){
          Object.defineProperty(WrappedComponent.prototype,"__"+
original_methods[i],Object.getOwnPropertyDescriptor(OriginalComponent.prototype,ori
ginal_methods[i]));
        }else{
```

```
Object.defineProperty(WrappedComponent.prototype,original_methods[i],Object.getOwn
PropertyDescriptor(OriginalComponent.prototype,original_methods[i]));
        }
      }
    }

    WrappedComponent.defaultClasses        = DefaultClasses;

    return WrappedComponent;
  }
};
```

Aunque en un principio pensamos en inyectar directamente los métodos del componente que extiende al original, después de algunas pruebas comprobamos que era mejor partir del componente extendido e inyectarle las funciones que no se han sobrescrito del componente original, ya que así evitaríamos problemas con el estado del componente, además de poder añadir variables al estado de este.

Este código comienza con la declaración de un componente que partirá siendo una copia del componente extendido. Después utilizando la función "Object.getOwnPropertyNames" obtenemos los nombres de las funciones tanto del componente que extiende como del componente original. Estos nombres de funciones se utilizarán para hacer el emparejamiento de métodos, donde si encontramos un método del componente original que ya existiera en el componente que extiende este sería inyectado utilizando el prefijo "__", de manera que el método original siempre sea accesible en caso de que fuera necesario.

Una vez que detectemos si la función debe ser insertada utilizando el prefijo o no, hacemos uso de la función "Object.getOwnPropertyDescriptor" sobre la función original para obtener la descripción de esta y poder añadirla a la copia del componente extendido a través de la función "Object.defineProperty".

Finalizada la inyección de métodos, añadimos a la copia las clases importadas en el componente, las cuales pueden ser nuevas o cambios de las originales, y devolvemos el componente. Con esto podemos conseguir extender cualquier componente original.

Un ejemplo de componente extendido sencillo sería el siguiente, que sustituye el logo original:

```
import React, { Component } from 'react';
import PropTypes from 'prop-types';

import logo from './logo.svg';

import defaultClasses from "./logo.css";

import {pluginBuilder} from '../../plugin';

class Logo extends Component {
```

```
    render() {
        const { height, classes } = this.props;
        return (
            <img
                className={defaultClasses.logo}
                src={logo}
                alt="main-logo"
                title="Logo"
            />
        );
    }
}

export default pluginBuilder(Logo, {});
```

Como se puede comprobar únicamente sustituimos el render del componente exportándolo a través de nuestra función contenida en "plugin.js" con dos parámetros, primero la clase y segundo las clases necesarias, en este caso no se sobrescribiría ninguna ya que utilizamos las clases CSS del componente original. En el caso que quisiéramos sobrescribir alguna clase esta sería nombrada igual que la original, se importaría el archivo CSS en el componente y se pasaría como segundo parámetro de la función "plugin".

Un ejemplo de cómo quedaría la extensión sería la siguiente, en la que se muestra el *header* y *footer* modificados acorde al estilo de la página de SierraBravaCarp:

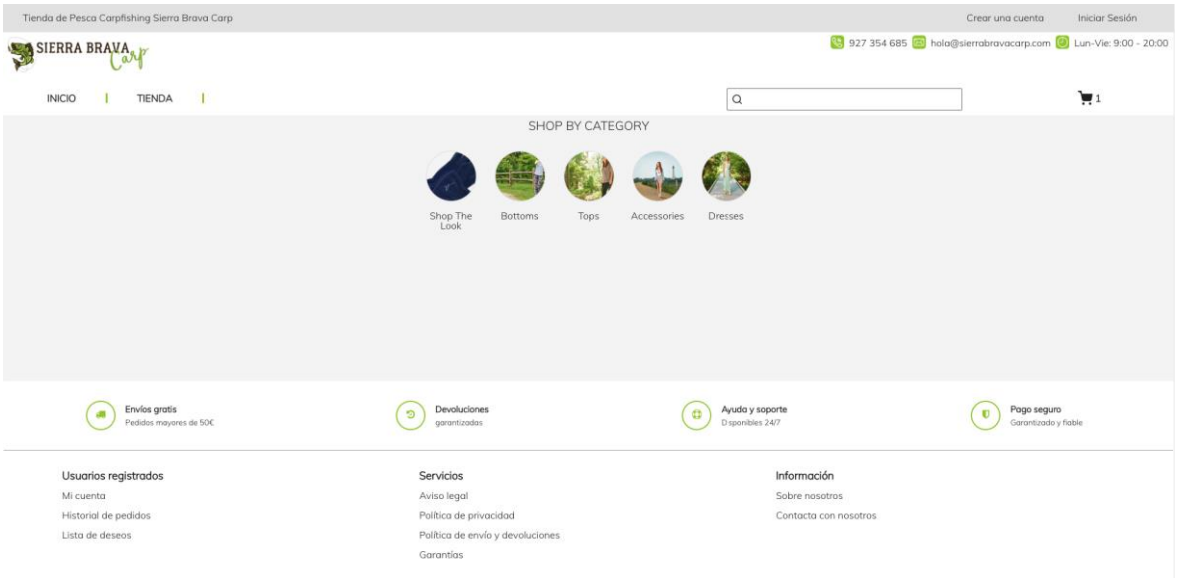

Figura 4-1: *Header* y *footer* modificados utilizando la extensibilidad desarrollada.

4.2 Adaptación de módulos de Magento con GraphQL

Debido a la extensibilidad desarrollada gran parte del *frontend* puede ser cambiado con poco esfuerzo de desarrollo, pero aún es necesario depender de ciertos módulos para completar el *frontend*. Uno de ellos es el llamado "MegaMenus" de CleverSoft.

Este módulo se usa para definir la estructura que tendrá el menú de nuestra página web de una manera muy sencilla desde el *backend* de la aplicación y esto es importante debido a que los menús pueden incluir links temporales, como una promoción temporal, y por tanto no es una buena práctica definir su estructura en código. Con este módulo se da la posibilidad al cliente de añadir o quitar elementos con una interfaz muy sencilla:

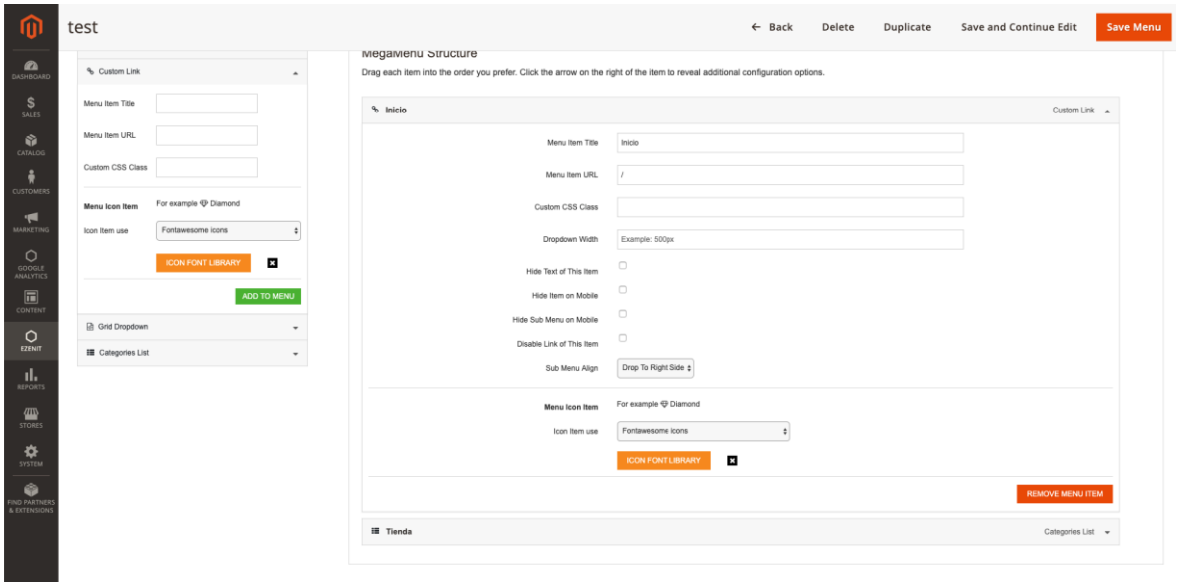

Figura 4-2: Vista de edición de elementos de "MegaMenus" en el administrador de Magento 2.

Como podemos ver la cantidad de personalización que se puede obtener con este módulo es muy grande, aunque en nuestro caso solo afectará a los elementos que hay en el menú y su tipo, no en su estilo o funcionalidad como es el caso de "MegaMenus" con el *frontend* actual de Magento.

Para ello lo que decidimos hacer es en la carpeta "extensions" del *backend*, donde se encuentran los módulos que se pueden compartir entre proyectos, crear un módulo que se encargará de recibir una *query* de GraphQl y responder con los elementos de los menús. Para ello damos de alta un nuevo módulo en Magento, en nuestro caso llamado "module-megamenus-graph-ql", y creamos el archivo "schema.graphqls" con la *query* y el tipo que devuelve:

```
type Query {
    megamenus(): [Menu]
    @resolver(class: "Ezenit\\MegaMenusGraphQl\\Model\\Resolver\\MegaMenus")
}

type Menu {
  id: String
  item_type: String
  label: String
  url: String
  category_id: String
```

}

Hemos optado por devolver un array con los distintos ítems de cada menú, con el id, el tipo de elemento ya sea categoría o enlace, la etiqueta con la que se debe mostrar, la url del enlace o de la categoría y en el caso de que sea categoría el id de esta, siendo "null" en el caso de un link. Un ejemplo de respuesta de esta *query* sería:

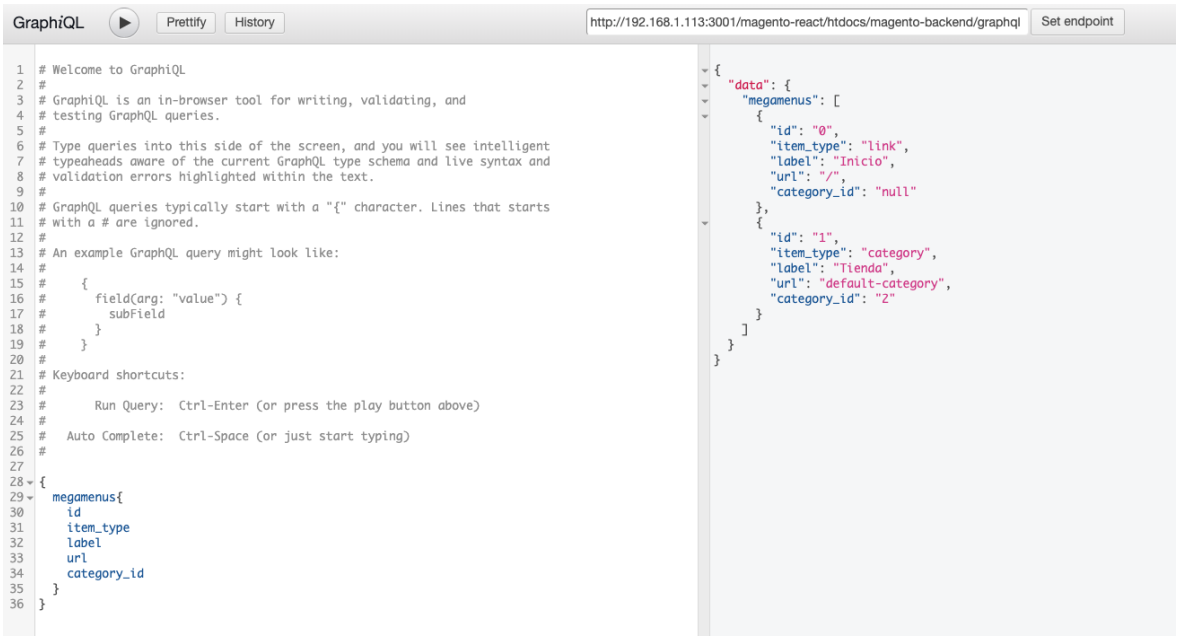

Figura 4-3: Ejemplo de respuesta de la *query* creada para el módulo "MegaMenus".

El resolver utiliza funciones del módulo "MegaMenus" para obtener la información introducida por el usuario en el administrador. Este módulo tiene una licencia privada y por tanto no podemos mostrar las funciones y el código de este.

Una vez obtenida la información del *backend* con esta *query* por lo que siempre dentro de nuestra carpeta "extensions" declarada en el paquete "venia-concept" creamos una carpeta denominada "module-megamenus" en la que introduciremos toda la lógica del menú. Nuestro módulo empieza con el componente que se exportará por defecto:

```
import React, {Component}       from 'react';
import { BrowserView,MobileView } from "react-device-detect";
import { Query }                from 'src/drivers';
import { Row, Col }             from 'react-bootstrap'
import MenuIcon                 from 'react-feather/dist/icons/menu';

import MegaMenuItem             from './MegaMenuItem';

import {pluginBuilder}          from '../../plugin';
import defaultClasses           from "./megamenu.css";
import megaMenu                 from '../queries/getMegaMenus.graphql';

import classify                 from 'src/classify';
```

```
const NavTrigger    = React.lazy(() => import('src/components/Header/navTrigger'));
const Icon          = React.lazy(() => import('src/components/Icon'));

class MegaMenu extends Component{
  render(){
    return (
      <Query query={megaMenu}>
       {({ loading, error, data }) => {
         const {megamenus=[]} = data;
         return (
           <div className={defaultClasses.ez_megamenu_content}>
            <BrowserView>
             {

               megamenus.map((data,index)=>{
                return (
                  <MegaMenuItem key={index} data={data} />
                )
               })
             }
            </BrowserView>
            <MobileView>
             <NavTrigger>
                <Icon src={MenuIcon} />
             </NavTrigger>
            </MobileView>
           </div>
         )
       }}
      </Query>

    )
  }
}

export default classify(defaultClasses)(MegaMenu);
```

En el tenemos una lógica de render que cambiará según la vista en la que estemos, en escritorio se mostrará un menú con elementos en horizontal, como el menú de la página SierraBravaCarp, y en móvil reutilizaremos el componente de "venia-concept", cambiando simplemente los elementos por los obtenidos a través de la *query* "megamenus".

Para la vista de escritorio hemos creado otro componente, denominado "MegaMenuItem" que se encargará de en función del tipo de elemento que se obtenga en la *query* como se va a mostrar:

```
import React,{Component}        from 'react';
import classify                 from 'src/classify';
import { Link,resourceUrl }     from 'src/drivers';
```

```
import defaultClasses           from "./megamenu.css";
import CategoryItem             from './CategoryItem';

import {CATEGORY_TYPE,LINK_TYPE}  from './constants';

class MegaMenuItem extends Component{

  render(){
    const {data:{label, item_type, url,category_id }} = this.props;
    const level = 0;
    return (
      <div className={defaultClasses[`level_${level}`]}>
       {
         (item_type===LINK_TYPE) &&

           <div className="item-link">
             <Link to={resourceUrl(url)}><span>{label}</span></Link>
           </div>
       }
       {
         (item_type===CATEGORY_TYPE) &&
           <CategoryItem       label={label}       categoryId={category_id}       url={url}
level={(level+1)} renderChildren={true}/>
       }
      </div>
    )
  }

}

export default classify(defaultClasses)(MegaMenuItem);
```

En el caso de los enlaces, simplemente utilizamos el componente de React “Link” para evitar el refresco de la página al acceder al enlace y en el caso de los elementos de tipo categoría renderizamos un componente, llamado “CategoryItem”, que se encarga de obtener la información de las categorías a través de la *query* ya existente en el proyecto PWA Studio denominada “getCategoryList” y recorrer de manera recursiva a los hijos para mostrar hasta dos niveles de hijos:

```
import React, {Component}          from 'react';
import { Query, Link,resourceUrl }  from 'src/drivers';
import classify                   from 'src/classify';
import {graphql}                    from 'react-apollo';
import defaultClasses               from "./megamenu.css";

import categoryList                 from '../../../queries/getCategoryList.graphql';

class CategoryItem extends Component{
  state = {
    showChildren: false
  }
```

```
showChildren = () =>{
  const { renderChildren } = this.props;
  if (renderChildren)
    this.setState({showChildren:true});
}
ocultChildren = (click=false)=>() =>{
  const { backOnClick, level } = this.props;
  if(backOnClick&&click){
    backOnClick();
  }
  this.setState({showChildren:false});
}

componentDidMount(){
  graphql(categoryList, {
    props:({data}) => ({
       children:data.children
    })
  })
}
render(){
  const { categoryId,url,label,level} = this.props;
  const { showChildren } = this.state;
  const renderChildren = (level < 2);

  return (
    <div  className={defaultClasses.item_category}  onMouseOver={this.showChildren}
onMouseLeave={this.ocultChildren()}>
      <div className="category_link">
        <Link                                    to={resourceUrl(`/${url}.html`)}
onClick={this.ocultChildren(true)}><span>{label}</span></Link>
      </div>
       {
         showChildren &&
         <Query query={categoryList} variables={{id:categoryId}}>
          {({ loading, error, data }) => {
            if(data.category){
              const { children=[]} = data.category;
              return (
                <div className={defaultClasses[`level_${level}`]}>
                 { children.map((children,index)=>{
                     const { id ,url_path ,name,children_count } =children;
                     const count = parseInt(children_count);
                     return(
                       <CategoryItem      key={index}      categoryId={id}      url={url_path}
label={name}      level={(level+1)}      renderChildren={(level<2      &&      count)}
backOnClick={this.ocultChildren(level>0)}/>
                     )
                   })
                 }
                </div>
```

```
            );
          }else{
            return null
          }
        }}
      </Query>
    }
  </div>
 )
}

}
```

export default classify(defaultClasses)(CategoryItem);

Finalmente, en nuestro *header* modificado llamamos al componente "MegaMenu" creado, obteniendo el siguiente resultado:

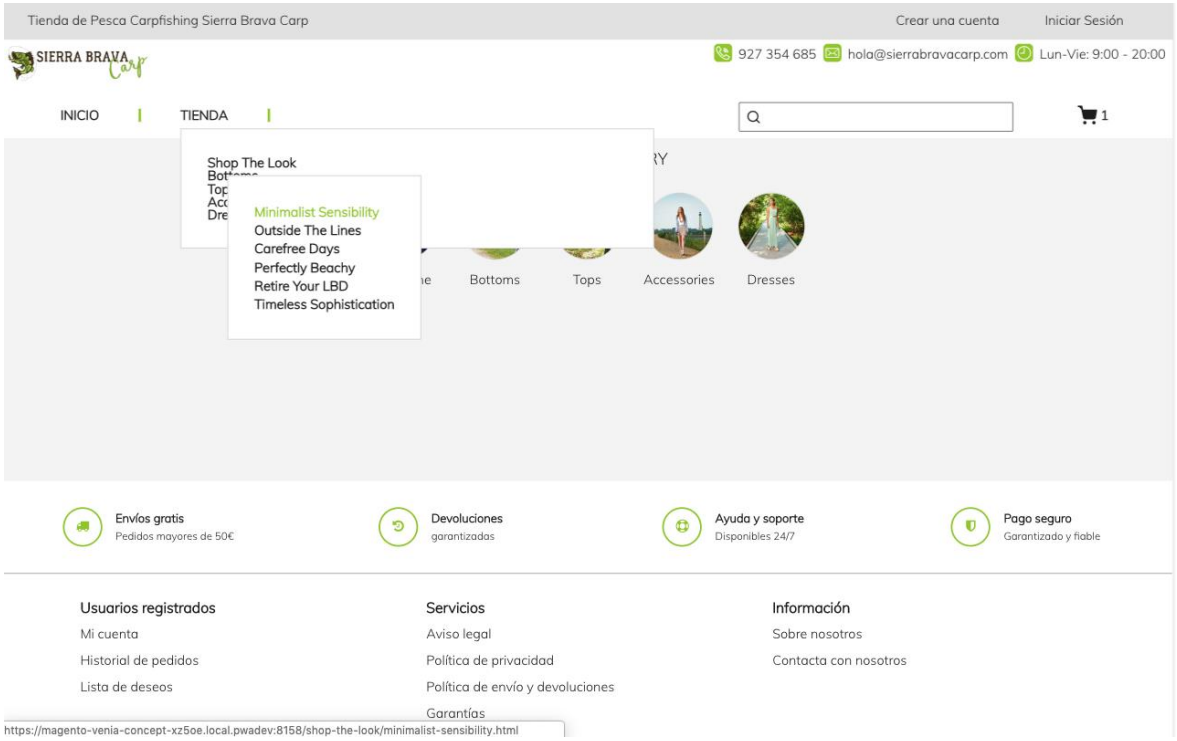

Figura 4-4: Resultado del menú utilizando el módulo "MegaMenus" adaptado al nuevo frontend.

5 Integración, pruebas y resultados

5.1 Pruebas y resultados

Para las pruebas se va a comparar la velocidad entre el *frontend* desarrollado con el que actualmente se encuentra en la página SierraBravaCarp, ambos en local y en modo desarrollador sin optimizaciones. Para ello haremos dos baterías, una sin limpiar cache y otra limpiando la cache del navegador, con tres páginas distintas, página principal, vista de categoría y vista de producto.

Las pruebas serán realizadas en un Mac Mini 2018 con las siguientes características:

- CPU: Intel Core i3 8100B
- Memoria RAM: 8GB DDR4 2667MHz
- Almacenamiento: 128GB Apple SSD PCI-Express

El navegador utilizado será Google Chrome, usando la pestaña "Performance" de las herramientas de desarrollador para medir el tiempo que tarda en mostrarse las distintas páginas.

Comenzamos las pruebas con el *frontend* desarrollado durante este trabajo de fin de grado, los tiempos mostrados son los promedios de cinco cargas en cada una de las páginas.

	Página principal	*Página vista de categoría*	*Página de vista de producto*
Sin cache de navegador	1.485,86 ms	1.817,56 ms	1.604,2 ms
Con cache de navegador	1.303,06 ms	1.763,9 ms	1.623,18 ms

Tabla 5-1: Tiempos de carga del *frontend* desarrollado

Y a continuación, las mismas pruebas realizadas con el *frontend* actual de Magento.

	Página principal	*Página vista de categoría*	*Página de vista de producto*
Sin cache de navegador	1.854,8 ms	1589,2 ms	2.179,6 ms
Con cache de navegador	1.563 ms	1280 ms	1.838,2 ms

Tabla 5-2: Tiempos de carga del *frontend* de Magento.

En este caso la diferencia entre hacer la carga con y sin cache es mayor que en el caso del *frontend* basado en React. Además, observamos que en la página de categoría el tiempo de carga es inferior en el *frontend* actual, lo cuál achacamos a que en el frontend de SierraBravaCarp la vista de categoría no muestra una galería con todas las imágenes del producto, mientras en nuestra implementación en React si se carga una galería con todas las imágenes del producto.

También observamos que la diferencia de tiempos con cache y sin cache en el caso del *frontend* de React es mínima, siendo lógico ya que parte del renderizado de React se realiza en el navegador, por lo que cache no afecta en exceso a los tiempos de carga. Por lo tanto, la ventaja de React como *frontend* es evidente, ya que en esta prueba en la que se refresca una página es la que menos aprovecha las ventajas de React, ya que hay que renderizar toda la página cada vez y no se puede renderizar solo el componente que cambia.

Donde más se nota la diferencia entre React y el *frontend* actual de Magento es en la navegación, siendo esta mucho más fluida y con menos interrupciones por carga al solo renderizar los componentes que cambian, como demuestra la siguiente prueba con los tiempos de renderizado, siempre haciendo un promedio de 5 mediciones, al pasar entre dos categorías y entre una categoría y un producto, dos acciones muy realizadas por parte de los consumidores en una página de comercio electrónico:

	Navegación entre dos categorías	*Navegación entre categoría y producto*
Frontend basado en React	435,67 ms	512,8 ms
Frontend actual de Magento	2845,45 ms	4312,23 ms

Tabla 5-3: Tiempos de renderizado en navegación

Aquí la diferencia es mucho mayor y es donde React verdaderamente demuestra sus virtudes. Renderizar solo los componentes que cambian junto a no tener que refrescar la página hacen que la navegación sea uno de los puntos fuertes de este *frontend*, sin olvidarnos de que gracias al desarrollo de PWA Studio podemos disfrutar de navegación sin conexión a internet, algo que sin duda ayudará al 70% en promedio de tráfico en dispositivos móviles que se genera en las webs gestionadas por la empresa.

Una vez visto los resultados de las pruebas de rendimiento, vamos a comprobar a través de algunas imágenes el resultado del desarrollo del *frontend* gracias a la extensibilidad.

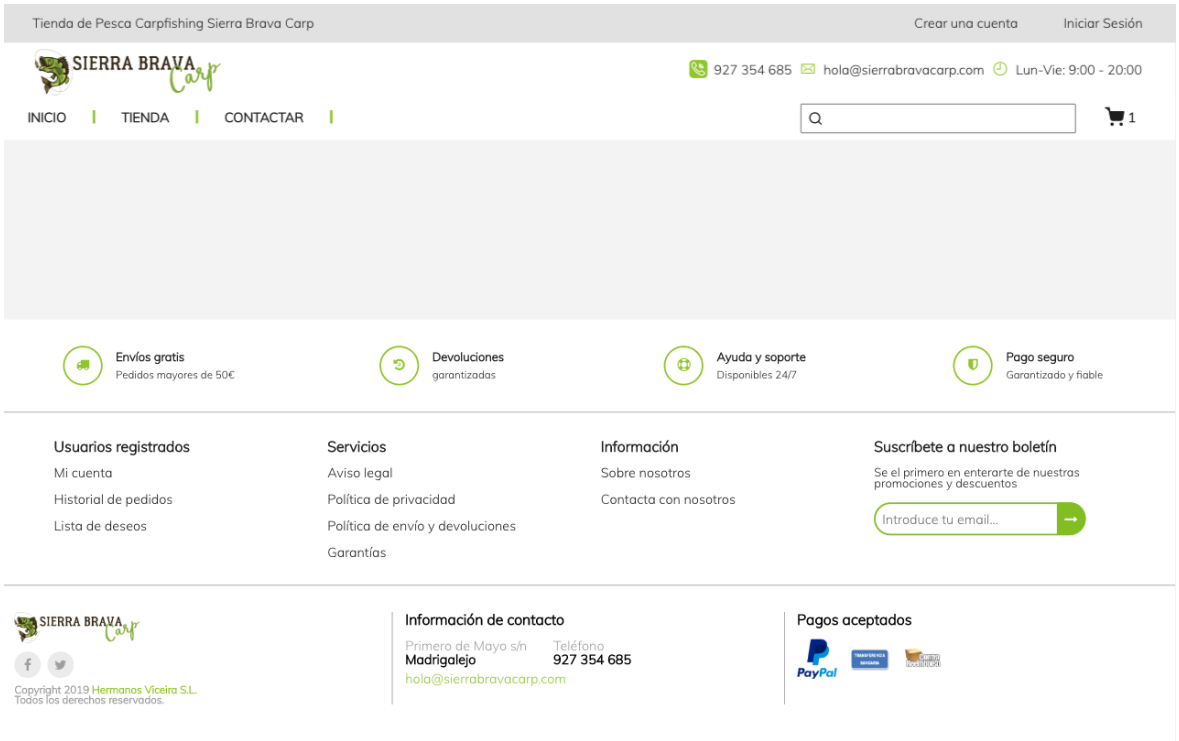

Figura 5-1: Versión final del *header* y el *footer* desarrollados.

Para empezar la página principal, con el *header* y *footer* de acuerdo a los criterios de la página actual de SierraBravaCarp, en la que se puede ver la integración del menú a través del módulo MegaMenus o la posibilidad de suscribirse al boletín informativo de la página.

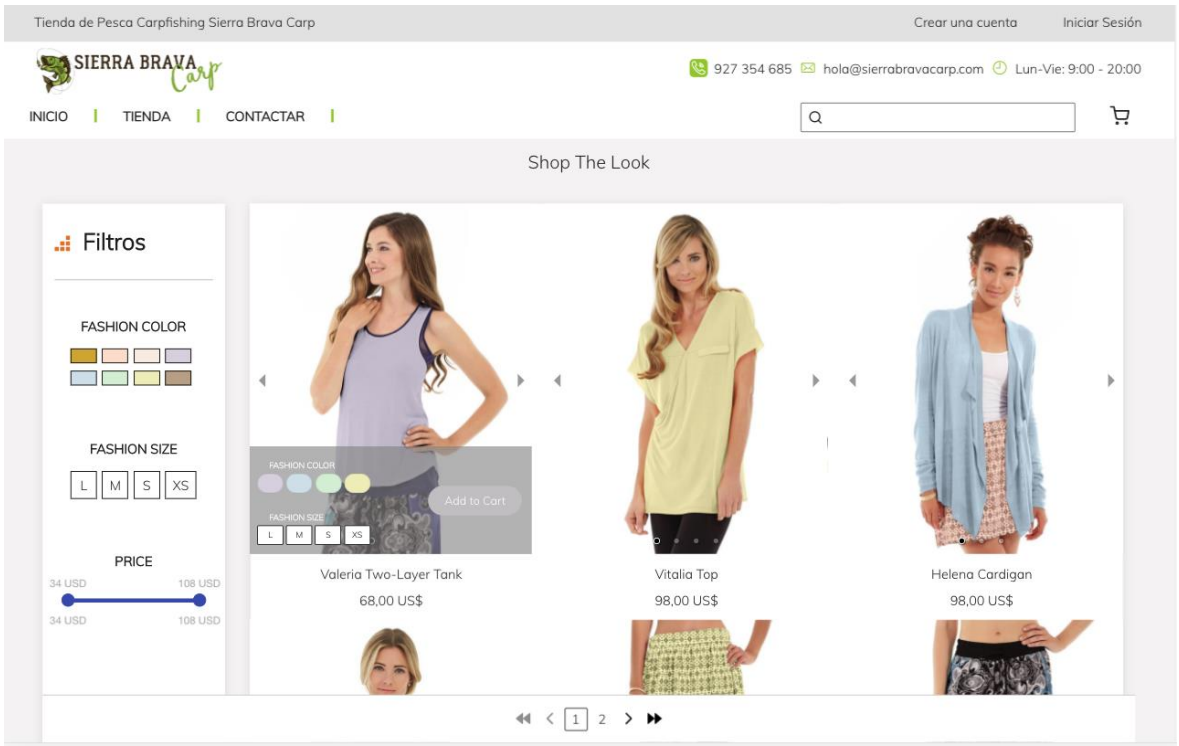

Figura 5-2: Vista de categoría mejorada con diversas características.

En la vista de categorías se han introducido diversas mejoras gracias a la extensibilidad. Se han añadido los filtros, recuperados desde el *backend* con GraphQL, además de una galería con todas las imágenes del producto junto con la compra rápida al pasar el ratón por encima de las imágenes, permitiendo al cliente añadir el producto sin tener tan siquiera que ir a la vista de producto.

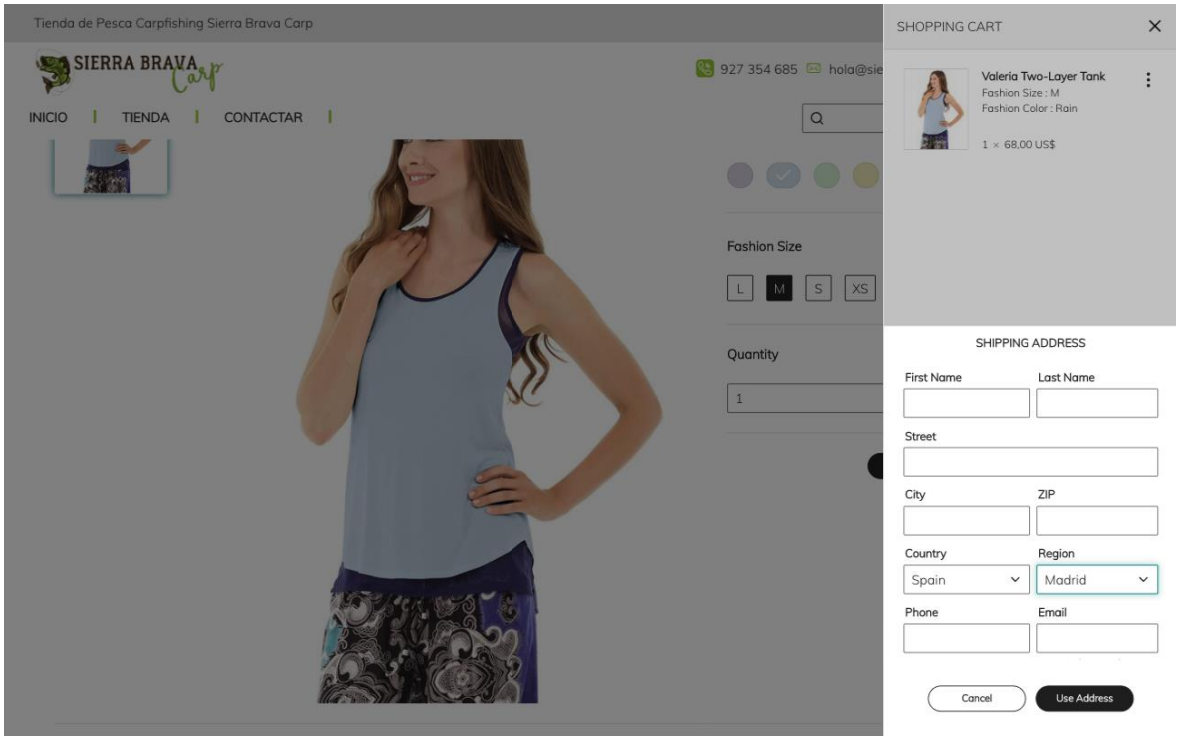

Figura 5-3: Mejora en el *checkout* para la introducción del país y región o provincia.

En el *checkout* se han realizado diversas mejoras, debido a la falta de estas en la implementación estándar de PWA Studio. Entre ellas estaban la imposibilidad de seleccionar el país y región o provincia de envío, siendo estas fijas a Estados Unidos, sin posibilidad de decidir desde el *backend* a donde se permiten envíos.

Los países son recuperados a través de la *query* ya implementada por parte de Magento para recuperar la lista de países a los que se permite el envío. Una vez seleccionado el país se procede a comprobar si este tiene regiones o provincias recuperándolas de nuevo con una *query*, esta vez añadida por nuestra parte, en la que se pasa el código correspondiente al país seleccionado y se devuelven las regiones o provincias correspondientes.

Ambas peticiones tienen en cuenta la configuración del *backend*, en la que se puede seleccionar los países o regiones a los que se permite el envío, dependiendo de las necesidades del cliente.

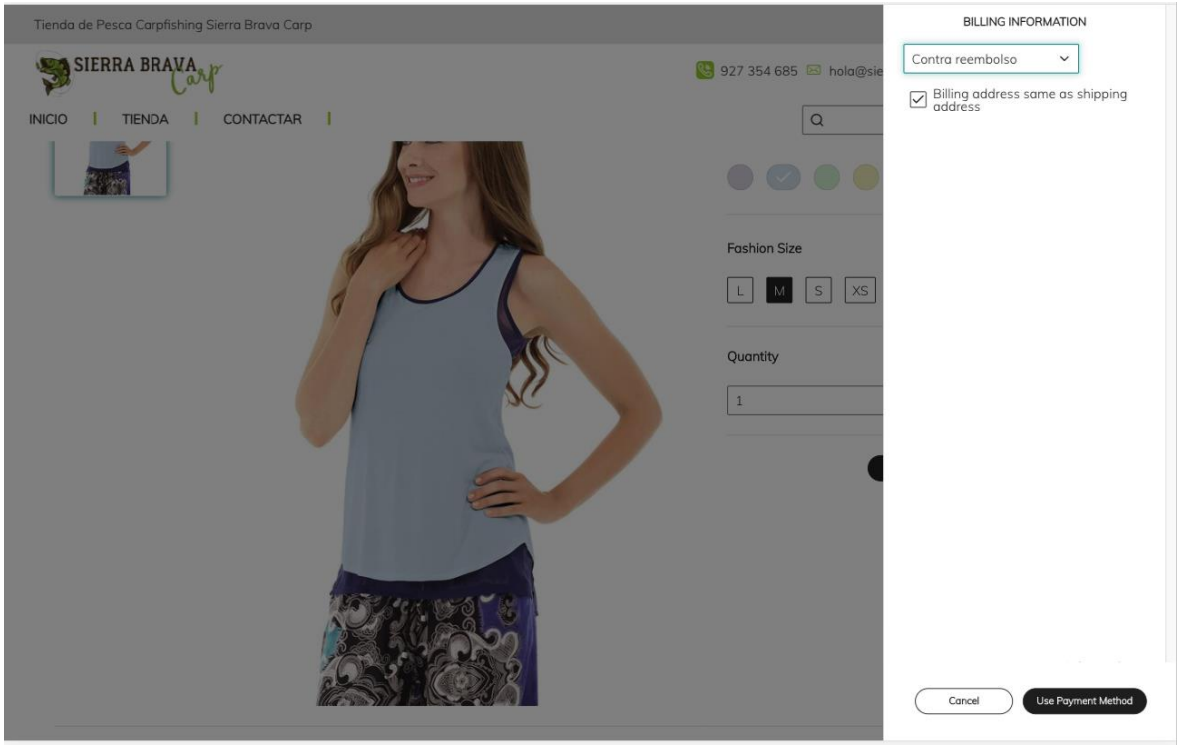

Figura 5-4: *Checkout* con diversos métodos de pago estándares de Magento.

También se añadió la posibilidad de aceptar diversos métodos de pago incluidos en Magento, siendo solo posible pagar con tarjeta a través de Braintree en el *frontend* estándar de PWA Studio.

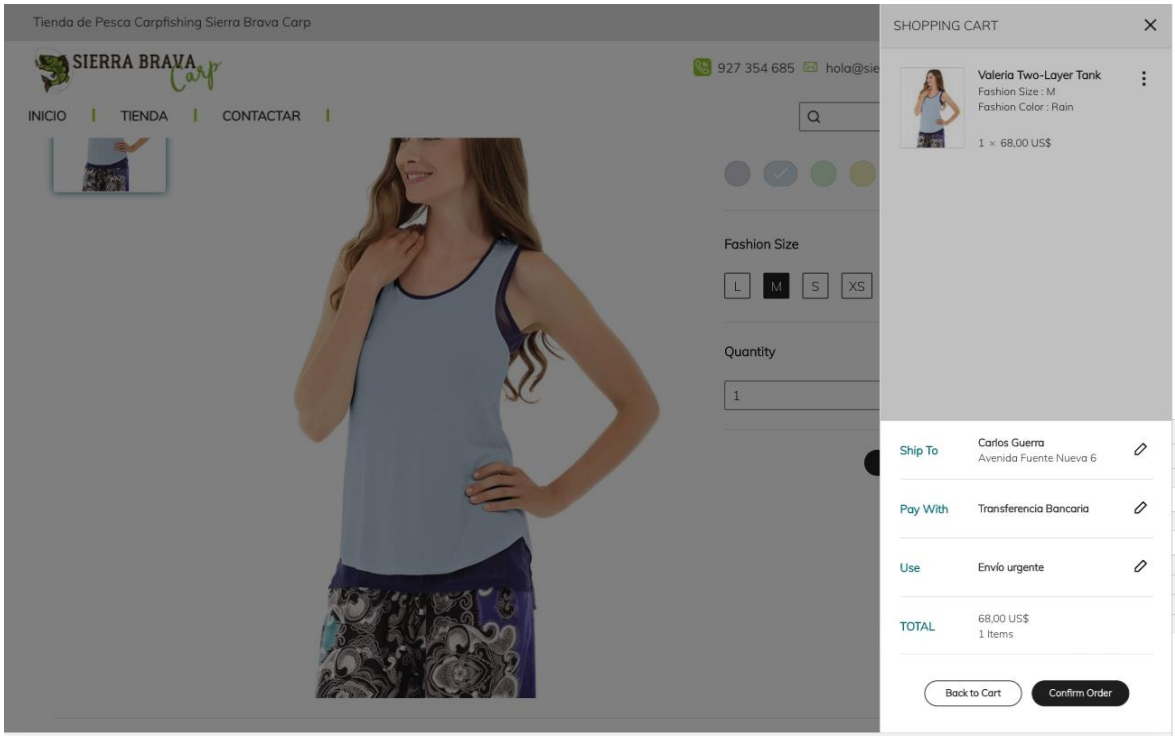

Figura 5-5: Vista general del *checkout* mostrando la información introducida por el usuario.

Toda la información introducida se muestra y transmite de igual manera que en el *checkout* anterior, siendo posible realizar pedidos, pero gracias a la extensibilidad realizada se han introducido varias mejoras que hacen aún más potente la experiencia de utilización de la página web, pudiendo estas ser reutilizadas para futuros desarrollos de manera muy sencilla e intuituva.

Además, el desarrollo y mantenimiento del código es más sencillo teniendo muy definidos los componentes y solo teniendo que modificar o crear archivos JavaScript y CSS, por los HTML, PHTML, LESS, XML y JavaScript del *frontend* actual.

6 Conclusiones y trabajo futuro

6.1 Conclusiones

Como hemos podido comprobar React tiene muchas ventajas sobre el *frontend* actual que presenta Magento y no es casualidad que estén desarrollando un proyecto el cual usa esta tecnología. La velocidad es superior, incluso sin una cache, y la facilidad con la que se puede dividir en componentes los distintos elementos hacen que el desarrollo del *frontend* sea más intuitivo que el actual.

Adaptarnos a este proyecto nos ha permitido no solo tener un *frontend* desarrollado en React, si no que tenemos acceso a las funciones propias de una aplicación web progresiva, como notificaciones o integración con sistemas operativos móviles.

El desarrollo llevado a cabo en este trabajo de fin de grado, consiguiendo extender un *frontend* básico para adaptarlo a distintos temas, otorga una facilidad en el desarrollo de proyectos al no tener que empezar de cero sino con una base que funciona, mejorando aún más las características del proyecto y sus posibilidades de cara a futuros proyectos en los que se piensa utilizar lo desarrollado durante este TFG.

También hemos asegurado la compatibilidad con versiones futuras del proyecto PWA Studio al no alterar la estructura en exceso, lo cual nos permitirá acceder a futuras actualizaciones de este proyecto y poder retroalimentarnos del trabajo desarrollado por la comunidad que en un futuro se adaptará a este nuevo *frontend*.

6.2 Trabajo futuro

Una de las cosas a mejorar de cara al futuro sería la forma en la que se establece que módulo sobrescribe a cuál. Creemos que este proceso se puede hacer más transparente, sin tener que establecer un *mapping* de componentes para determinar cual es el componente extensible.

Ya hemos valorado la posibilidad de que esto se haga de manera automática, algo similar a los que hace Magento en su carpeta “design”, donde simplemente estableciendo el nombre del módulo y nombrando los ficheros de igual manera, estos tienen preferencia sobre los propios de los módulos. Esta idea nos llevaría a tener en la carpeta “extensions” una estructura similar a la que presenta la carpeta “components” de manera que antes de devolver el componente al render se busque su componente extendido automáticamente.

Por otro lado, aunque los módulos de CSS tienen ventajas, creemos que utilizar LESS en su lugar aportaría aún más ventajas y ficheros con código más legible y fácil de mantener. Aunque durante el desarrollo se intento utilizar LESS para los componentes extendidos no se pudo debido a la estructura que presenta el “webpack” del proyecto, dejando esta idea como mejora para un futuro cuando el desarrollo del proyecto PWA Studio este más avanzado para no interferir con las posibles actualizaciones que se pueden implementar

con facilidad al no tocar la estructura en exceso, como se comentaba en la sección de conclusiones.

Finalmente, con la experiencia y desarrollos conseguidos con este trabajo de fin de grado, en el laboratorio de la empresa se está comenzando el desarrollo de un *frontend* unificado de React para diversas plataformas de comercio electrónico. Utilizando GraphQL para comunicar el *frontend* y el *backend* de manera transparente y con definición de estructuras y *querys* unificadas se pretende tener un diseño muy modulable y en el que no se depende del *backend*, pudiendo adaptar un mismo *frontend* a diversos *frameworks* de comercio electrónico como Prestashop o Shopify, además de Magento.

Referencias

[1] “Magento theme structure”, Magento Developer Documentation, version 2.3 https://devdocs.magento.com/guides/v2.3/frontend-dev-guide/themes/theme-structure.html

[2] “What is a Progressive Web App”, Magento PWA Documentation https://magento-research.github.io/pwa-studio/technologies/overview/

[3] “What is the Magento PWA Studio project” Magento PWA Documentation https://magento-research.github.io/pwa-studio/technologies/overview/

[4] “Composition vs Inheritance”, React Documentation https://reactjs.org/docs/composition-vs-inheritance.html#so-what-about-inheritance

Glosario

API	Application Programming Interface
Magento	Framework de código abierto escrito en PHP para el desarrollo de páginas web de comercio electrónico.
ReactJS	Librería de JavaScript para el desarrollo de interfaces de usuario.
GraphQL	Lenguaje que permite la recuperación o manipulación de información de un *backend* con independencia del *framework* o lenguaje de este.